VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM KONVERZÍ KONEČNÝCH AUTOMATŮ

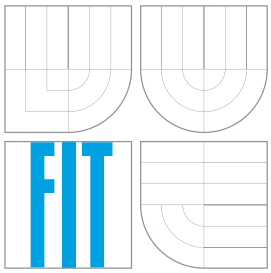
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VOJTĚCH ŠTOREK

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEMONSTRAČNÍ PROGRAM KONVERZÍ KONEČNÝCH AUTOMATŮ

DEMONSTRATION PROGRAM OF CONVERTION OF FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VOJTĚCH ŠTOREK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ROMAN LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2008

Zadání a licenční smlouva je uvedena v archivním výtisku uloženém v knihovně FIT VUT v Brně.

Abstrakt

Cílem praktické části této práce je vytvořit program, který aplikuje teorii konečných automatů v praxi a dělá tak tuto teorii snadější k pochopení. Program umožňuje snadné vytváření konečných automatů, nad nimiž následně jednoduše, ale hlavně didakticky demostruje základní teoretické znalosti, jako jsou převody na speciální typy konečných automatů, ilustrace činnosti nebo vyjádření konečného automatu formou zdrojového souboru v jazyce C. V této technické zprávě se pokusím popsat, jak jsou jednotlivé klíčové částí aplikace implementovány.

Klíčová slova

Konečný automat, převod, ilustrace činnosti, generování kódu, grafické uživatelské rozhraní, c++, wxWidgets.

Abstract

The main goal of practical part of this work is to create a program, which applies finite automata theory in practice and makes this theory easier to understand. Program is capable to easily create finite automata and make didactic demostration of basic theoretical knowledges such as coversion to special types of finite automata, function illustration or representation of finite automata in form of C language source code.
In this technical report I will try to describe, how are individual key parts of the application implemented.

Keywords

Finite automata, conversion, function illustration, code generation, graphical user interface, c++, wxWidgets.

Citace

Vojtěch Štorek: Demonstrační program konverzí konečných automatů, bakalářská práce, Brno, FIT VUT v Brně, 2008

Demonstrační program konverzí konečných automatů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Romana Lukáše, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Vojtěch Štorek
7. května 2008

Poděkování

Chtěl bych poděkovat svému vedoucímu, panu Ing. Romanu Lukášovi, Ph.D., za umožnění zpracování tohoto tématu.

Obsah

Úvod

Předmětem této bakalářské práce jsou konečné automaty. Kolem konečných automatů existuje mnoho teorie, současně ovšem platí, že člověk dokáže věci mnohem lépe chápat, pokud jsou názorně předvedeny v praxi. Přesně o to jde v této práci — demonstrovat teorii konečných automatů prakticky.

Praktickou částí bakalářské práce bylo vytvořit program, který demonstruje teorii konečných automatů, zatímco tato technická zpráva se pokouší tento program popsat. Postup vysvětlování v této zprávě bude takový, že ve většině kapitol bude nejdříve daná část popsána teoreticky, pak se k ní přidá popis její implementace.

Je nutno poznamenat, že pokud v této práci mluvíme o konečných automatech, jedná se o konečné automaty ve vztahu k formálním jazykům jako model pro regulární jazyky. Teorii k těmto konečným automatům a to, jak byly implementovány v aplikaci, se věnuje kapitola 1.

Jednou z nejtěžších teoretických pasáží konečných automatů jsou převody konečných automatů na speciální typy. Speciálními typy rozumíme např. konečný automat bez ϵ-přechodů nebo deterministický konečný automat, atd. Existuje několik variant automatů a pro každou variantu existuje speciální algoritmus. Tyto algoritmy a jejich implementace popisuje kapitola 2.

Kapitola 3 popisuje funkci programu, v níž je možné vyzkoušet činnost konečného automatu. Pro zadaný vstupní řetězec jsou hledána a aplikována pravidla a výsledkem je, zda-li konečný automat přijímá nebo nepřijímá tento řetězec.

Exportu konečného automatu, nebo-li vyjádření konečného automatu jinou formou, se věnuje kapitola 4. Jinou formou se v tomto případě myslí jednak zdrojový kód v jazyce C, který po překladu a spuštění pracuje přesně jako vytvořený konečný automat, jednak obrázek s daným konečným automatem.

V kapitole 5 jsou popsány další důležité funkce aplikace, jako jsou automatické rozmístění stavů, formát XML souboru pro ukládání konečných automatů, podpora více jazyků, atd.

Celkové shrnutí práce je k dispozici v předposlední kapitole 6, kde jsou diskutovány výhody a nevýhody projektu a jeho další možná rozšíření.

Praktická část bakalářské práce je psána v jazyce C++ spolu s knihovnou wxWidgets [8]. Deklarace, definice a úseky kódu, které se objeví v této práci, budou proto syntaxí odpovídat právě tomuto jazyku, v ostatních případech se může také jednat o pseudokód.

Kapitola 1

Konečný automat

V této kapitole jsou popsány formální definice týkající se konečných automatů. Také jsou zde uvedeny dvě reprezentace konečných automatů v aplikaci — matematická a grafická. Dva modely jednoho prvku se mohou zdát zbytečné, ale uvidíme, že to má svůj důvod a že tyto dva modely nejsou zase tak moc odlišné.

1.1 Teorie konečných automatů

Mluvíme-li o konečných automatech, pak máme na mysli konečné automaty jako modely pro regulární jazyky. Než si jej nadefinujeme, potřebujeme znát pojem *abeceda*.

Definice 1.1. *Abeceda* je konečná, neprázdná množina elementů, které nazýváme symboly.

Nyní již můžeme přikročit k pojmu *konečný automat*.

Definice 1.2. *Konečný automat* je pětice:

$$M = (Q, \Sigma, R, s, F), \text{kde}$$

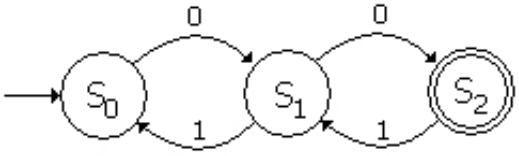
- Q je konečná množina stavů,
- Σ je vstupní abeceda,
- R je konečná množina pravidel tvaru:

  $$pa \rightarrow q^{1},$$

  kde $p, q \in Q, a \in \Sigma \cup \{\epsilon^{2}\}$,
- $s \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina koncových stavů.

Na obrázku 1.1 je zobrazeno grafické znázornění konečného automatu. Tomuto automatu by podle definice 1.2 odpovídal následující popis (uvádím jen obsah jednotlivých prvků konečného automatu):

[1] $pa \rightarrow q$ znamená, že při přečtení a M udělá přechod z p do q.
[2] ϵ značí tzv. prázdný řetězec = neobsahuje žádný symbol.



Obrázek 1.1: Grafická reprezentace konečného automatu

	0	1	ϵ
S_0	$\{S_1\}$		
S_1	$\{S_2\}$	$\{S_0\}$	
$\underline{S_2}$	$\{S_1\}$		

Tabulka 1.1: Tabulková reprezentace konečného automatu

- $Q = \{S_0, S_1, S_2\}, \Sigma = \{0, 1\}, R = \{S_0 0 \rightarrow S_1, S_1 0 \rightarrow S_2, S_2 1 \rightarrow S_1, S_1 1 \rightarrow S_0\}, s = S_0, F = \{S_2\}$.

Stavy se tedy znázorňují jako kolečka s názvem stavu uvnitř, počáteční stav je označen šipkou (ta jediná šipka, která není umístěna mezi dva stavy), koncové stavy mají kolečko nakreslené dvojitou čarou. Pravidla jsou znázorněna jako křivky mezi dvěma stavy, symbol pravidla je umístěn na této křivce.

Konečné automaty mohou být reprezentovány i jiným způsobem. Tabulka 1.1 popisuje odpovídající konečný automat z obrázku 1.1. Ve sloupcích jsou prvky z $\Sigma \cup \{\epsilon\}$, v řádcích stavy z Q. Na prvním řádku je počáteční stav. Koncové stavy automatu jsou podtrženy. Platí, že průsečík stavu p a symbolu a obsahuje množinu stavů $q \in Q$, pro které platí $pa \rightarrow q \in R$.

Další informace o konečných automatech lze najít v knize A. Meduny [7], odkud byly převzaty i tyto definice. Čerpáno bylo také z přednášek kurzu Formální jazyky a překladače na FIT VUT v Brně [5].

1.2 Modely konečného automatu

Teď si představte situaci, že chcete informace o konečném automatu uchovávat v programu. Pokud bychom chtěli co nejvěrnější podobu podle definice 1.2, pak bychom deklarovali datový typ pro množinu symbolů, tedy abecedu, datový typ pro množinu stavů a datový typ pro množinu pravidel (definici symbolů, stavů a pravidel samotných zatím nebudeme brát v úvahu). Přesně tomuto návrhu tedy odpovídá matematický model v následující části práce.

Nicméně program má být názorný a dobře použitelný k praktickým ukázkám. Co se názornosti týče, tak podle mého si každý udělal největší představu o tom, jaké složení má konečný automat (tím myslím, jak jsou spolu stavy propojeny a jaká pravidla směřují z jakého stavu a s jakým symbolem do jakého stavu) z grafické reprezentace, tedy z obrázku 1.1. Zde je nejvíce viditelné, jak asi bude tento daný automat fungovat. Grafický model konečného automatu tedy nese mimo informací o abecedě, stavech a pravidlech navíc informace, které jsou potřeba pro tuto reprezentaci.

Oba výše uvedené modely jsou mezi sebou jednouše převeditelné. Převádíme-li však matematický model na grafický, nemáme k dispozici žádné informace o rozmístění stavů, příp. pravidel. Tento problém je dále popsán v kapitole 5.3.

Pro oba modely je společný typ pro abecedu, který je definován jako množina znaků (šablonová třída pro množinu — *set* z knihovny STL jazyka C++).

1.2.1 Matematický model

Matematický model je tedy věrnou kopii formálně definovaného konečného automatu (s výjimkou množiny koncových stavů, více v následující části).

Stavy

Třída pro stav nese v matematickém modelu dvě informace. První je název stavu. Na obrázku 1.1 je názvem stavu například řetězec *S0* (indexy bohužel nepřipadají v úvahu, proto je název zapsán takto). Druhou položkou, kterou třída pro stav nese, je informace o tom, zda-li je stav koncový. Koncové stavy tedy nejsou ukládány jako množina stavů konečného automatu, která je podmnožinou množiny stavů, ale každý stav nese informaci o své koncovosti v sobě.

Pravidla

Pravidla jsou modelována opět tak, aby odpovídala definici konečného automatu (definice 1.2). Skládají se ze dvou stavů, kde jeden je zdrojový (na levé straně pravidla) a druhý cílový (na pravé straně pravidla) a ze symbolu.

Konečný automat

Třída konečného automatu podle definice 1.2 obsahuje:

- množinu stavů — deklarována jako množina objektů typu stav (třída *set*),
- vstupní abecedu — její definice byla uvedena dříve,
- množinu pravidel — deklarována jako množina pravidel objektů typu pravidlo (třída *set*),
- počáteční stav — řetězec s názvem počátečního stavu,
- a množina koncových stavů — ta není součástí automatu, jak již bylo řečeno dříve, ale tato informace je přímo součástí stavů.

Množina stavů a pravidel obsahuje pouze jedinečné prvky. U stavů se jedinečnost rozlišuje na základě názvů stavů. Porovnání dvou stavů se nezjišťuje z jejich přesné řetězcové podoby, ale názvy jsou nejdřív vhodně upraveny a až poté porovnány. V první řadě je důležité říct, že název stavu smí být tvořen pouze alfanumerickými znaky (nesmí začínat číslicí, znaky pouze z ASCII), podtržítkem a znaky {}, (slovy: otevírací a uzavírací složená závorka a čárka). Tyto znaky se do názvů stavů totiž vloudí při převodech konečných automatů, kdy jeden stav zastupuje množinu stavů. Při porovnání se v názvu stavu znaky {}, nahradí znakem podtržítka a malé znaky jsou převedeny na velké.

Příklad 1.1. _A_ == {a} A_b == a,B a != {a}

Důvodem, proč jsou názvy stavů takto ošetřeny, je omezení názvů identifikátorů v jazyce C++. V kapitole 4.1.1 budou důvody vysvětleny přesněji.

Jedinečnost pravidel je zajištěna porovnáním jednotlivých prvků, kterými je pravidlo určeno. Jedná se o zdrojový stav, čtený symbol a cílový stav. Názvy obou stavů jsou při porovnávání upraveny, jak je popsáno v předchozích odstavcích.

Příklad 1.2. $pa \rightarrow q1 == pb \rightarrow q2$ se porovná jako $P == P$ **and** $Q1 == Q2$ **and** $a == b$.

1.2.2 Grafický model

Jak už bylo řečeno v úvodu této podkapitoly, grafický model nese oproti matematickému modelu navíc informace důležité pro vykreslování. Mezi těmito dvěma modely je však jistá spojitost.

Stavy

Třída pro stav nese informace nejen o názvu a příznaku, zda-li je daný stav koncový, ale také informace o svých souřadnicích a rozměrech a grafických prvcích jako barva písma a pozadí.

Pravidla

Třída pro pravidla je v případě grafického modelu komplikovanější. Deklarována je jedna třída pro pravidla, která určuje rozhraní, které budou všechny odvozené třídy implementovat. Toto rozhraní přebírají tři třídy pro tři typy pravidel.

1. Prvním typem pravidel je normální pravidlo. Jedná se o třídu, která obdrží informace o tom, z jakého stavu směřuje do jakého stavu a s jakým symbolem. Na základě souřadnic stavů vytvoří křivku. Mohou být specifikovány další body, podle nichž bude křivka tvarována. Navíc se na konci křivky u cílového stavu vytvoří šipka, která naznačuje směr pravidla. Symbol pravidla se je vytištěn do středu křivky těsně nad ni.

2. Druhou třídou pro pravidlo je zástupné pravidlo. To se navenek chová stejně jako předchozí uvedená třída, je mezi nimi však velký rozdíl. Této třídě jsou také předány tři klíčové informace (zdrojový a cílový stav a symbol). Čtvrtou informací je objekt třídy normálního pravidla, na kterém bude parazitovat. Takto vytvořený objekt se „nalepí“ na objekt, který mu byl předán a všechny své operace přeposílá jemu. Současně také svůj symbol přidá k symbolu normálního pravidla, aby byl zobrazován.

   Tím je zajištěn menší počet náročných operací jako vykreslování, přesouvání, přibližování/oddalování, zjišťování, jestli se nad pravidlem nenachází kurzor myši, atd. Každé pravidlo je samostatným objektem, jak je to definováno v konečných automatech, i když pro všechna pravidla $pa \rightarrow q, \forall a \in \Sigma$ je vykreslována pouze jedna křivka.

3. Poslední třídou už není ani tak třída pro pravidlo, ale s pravidlem má společné to, že se jedná o křivku vedoucí do stavu s šipkou na konci. Jedná se o označení počátečního stavu. I když by se zdálo zbytečné, aby tyto třídy měly cokoli společné, opak je pravdou. V mém případě se vlastně jedná o pravidlo, které nemá zdrojový stav (místo toho je specifikován pouze jeden bod) a nemá žádný čtený symbol. Jakmile se na toto

speciální pravidlo podíváme z tohoto úhlu, pak máme všechny operace vyjmenované v předchozím odstavci (vykreslování, přesouvání, ...) ujednoceny na všechny křivky v konečném automatu.

Pro modelování křivek jsou použity NURBS[3], o kterých se můžete dozvědět např. v knize J. Žáry [10]. V mé práci jsem však využil již naimplementované NURBS křivky z kurzu Základy počítačové grafiky [6] na FIT VUT v Brně.

Konečný automat

Konečný automat obsahuje mimo informace o stavech a pravidlech také detaily pro grafické znázornění.

Ale i stavy a pravidla jsou ukládány jinak než u matematického modelu. Nejedná se již o množiny, ale mapy (hashovací tabulky - šablonová třída *map* z knihovny STL jazyka C++).

Mapa má jako index objekt matematického modelu a jako hodnotu objekt grafického modelu. Právě zde se vytváří spojitost těchto dvou modelů. Důvodem, proč jsou zvoleny mapy namísto množin, je jednodušší přístup k prvkům přímo pomocí indexu. Jedinečnost prvků je přitom stále zachována, protože na jednom indexu smí být pouze jedna hodnota.

Tomuto přístupu předcházel ještě jiný způsob ukládání. Nejdříve byly stavy a pravidla ukládána do dynamického pole (*vector* z STL). Operace, které třída konečného automatu poskytovala, byly prováděny na základě identifikátoru (např. u operace pro přesun stavu byla volána metoda konečného automatu, které byl předán identifikátor stavu). Tento identifikátor obdržel každý stav a pravidlo při vytvoření. Identifikátor navíc sloužil jako index do dynamického pole, aby byl urychlen přístup k prvkům. Pokud byl smazán stav někde uprostřed pole, pak se část pole posunula, aby nevznikala volná místa, čímž se identifikátory změnily.

Problém vznikl s příchodem tříd pro příkazy konečného automatu, který je implementován z důvodu podpory akcí, které lze provést a odvolat zpět (tzv. Zpět a Znovu, Undo a Redo) — více k těmto operacím v kapitole 5.4. Pokud by byly za běhu aplikace provedeny dva příkazy, např. přesun stavu a následné smazání stavu, pak obě operace pracovaly s identifikátorem stavu. Průběh těchto operací vypadá následovně:

1. Stav s identifikátorem x přesunut.
2. Stav s identifikátorem x smazán.

Nyní si představte, že chcete tyto operace vzít zpět, resp. provést Undo. Toho je docíleno tak, že jsou k těmto operacím zavolány inverzní operace, tedy k odstranění je zavoláno přidání a k přesunu je zavolán přesun s negovanými souřadnicemi.

1. Přidán stav s identifikátorem y (který obdrží nově při vytvoření)
2. Stav s identifikátorem x přesunut.

Příkaz přesunu nemá žádné ponětí, že identifikátor stavu, nad nímž byl proveden přesun, už nemá tento identifikátor, ale byl mu vygenerován nový. Operace přesunu bude vykonána nad nesprávným stavem.

[3] Nonuniform rational B-spline

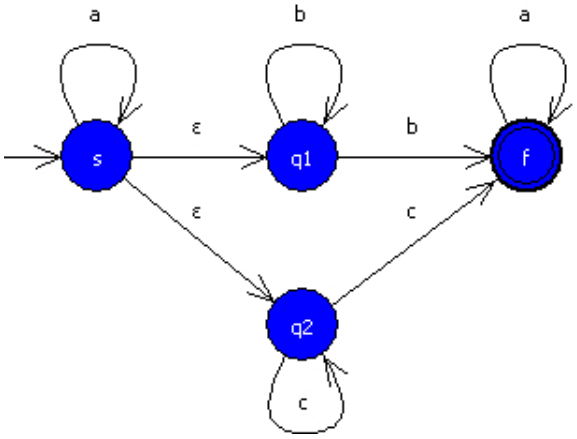
Tento případ by se ještě mohl zdát řešitelný, horší je ale případ, kdy by byla operace prováděna nad skupinou stavů. Např. příkazu pro mazání stavů by byly předány tři identifikátory třech stavů. Smazal by se první, mezitím by se však mohly identifikátory zbylých dvou změnit a smazány by byly úplně jiné stavy, příp. by se přistupovalo za meze pole.

Tento přístup tedy nepřipadal v úvahu, a proto byly jako indexy zvoleny objekty třídy matematického modelu. V podstatě se jedná o indexování na základě řetězce, na nějž jsou stavy a pravidla matematického modelu převáděny (způsobem uvedeným v 1.2.1). Pokud bude za těchto okolností proveden stejný sled operací jako v předchozím příkladu, budou všechny operace pracovat s objektem matematického modelu, jehož řetězcová reprezentace bude stejná, ať už je vytvořen či smazán, kolikrát chce.

Konečný automat má tedy jednotlivé elementy z definice 1.2 vytvořeny následovně:

- množina stavů — deklarována jako mapa s indexací pomocí stavů matematického modelu a s hodnotami ukazatel na stav,
- vstupní abeceda — její definice byla uvedena dříve,
- množina pravidel — deklarována jako mapa s indexací pomocí pravidel matematického modelu a s hodnotou ukazatel na bázovou třídu pro pravidlo,
- počáteční stav — objekt pro označení počátečního stavu (třetí typ tříd pro pravidla),
- a množina koncových stavů — opět není součástí automatu, ale tato informace je přímo součástí stavů.

Důvod, proč jsou u stavů a pravidel hodnotou mapy ukazatele, je využití polymorfizmu. U pravidel existují totiž tři typy tříd, podle nichž se objekty vyrábí. U stavů je to pouze z důvodu jednotnosti (když už jsou pravidla ukládána jako ukazatele, pak tak budou ukládány i stavy).



Obrázek 1.2: Aplikací vytvořený konečný automat

Na obrázku 1.2 je zobrazen konečný automat, který byl vytvořen pomocí aplikace. Sami by jste teď jistě dokázali vytvořit formální popis konečného automatu, proto jej nebudu uvádět.

Kapitola 2

Převody konečných automatů

Tato kapitola se věnuje převodům konečných automatů na jejich speciální typy. Speciální konečné automaty mají obecně lepší vlastnosti a bývají v praxi použitelnější. Konečný automat na obrázku 1.2 má hned několik vlastností, které jsou v praxi jen těžko použitelné (např. ϵ-přechody, nedeterminismus, atd.). Všechny vlastnosti a toho, jak jich dosáhnout, budou popsány v následujících podkapitolách.

Aplikace dokáže převádět vytvořené konečné automaty na následující speciální typy:

- Konečný automat bez ϵ-přechodů
- Deterministický konečný automat
- Konečný automat bez neukončujících stavů
- Dobře specifikovaný konečný automat
- Minimální konečný automat

Než se pustíme do popisu jednotlivých speciálních typů konečných automatů a toho, jak jich dosáhnout, je potřeba si trochu rozšířit teorii kolem konečných automatů.

Teorie a algoritmy zde uvedené byly opět převzaty z knihy A. Meduny [7] a z přednášek kurzu Formální jazyky a překladače [5].

2.1 Teorie

Potřebujeme znát pojem *ekvivalentní modely*. Výstupem všech dále zmíněných převodů (a jejich algoritmů) je totiž vždy model, který je ekvivalentní se svým vstupem. Nejprve si proto musíme zavést nové základní pojmy *řetězec* a *jazyk*.

Definice 2.1. Nechť Σ je abeceda.

1. ϵ je řetězec nad abecedou Σ
2. pokud x je řetězec nad Σ a $a \in \Sigma$, potom xa je řetězec nad abecedou Σ.

Definice 2.2. Nechť Σ^* značí množinu všech řetězců nad Σ. Každá podmnožina $L \subseteq \Sigma^*$ je *jazyk* nad Σ.

Nyní si rozšíříme teoretické znalosti konečných automatů. Nejprve si nadefinujeme *konfiguraci*, v níž využijeme definic 2.1 a 2.2. Poté můžeme přejít k definici *přechodu* (na který již bylo poukázáno v sekci Teorie konečných automatů, viz. 1.1) a *sekvenci přechodů*.

Definice 2.3. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. *Konfigurace* konečného automatu M je řetězec $\chi \in Q\Sigma^*$.

Definice 2.4. Nechť pax a qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, a \in \Sigma \cup \{\epsilon^1\}$ a $x \in \Sigma^*$. Nechť $r = pa \to q \in R$ je pravidlo. Potom M může provést *přechod* z pax do qx za použití r, zapsáno $pax \vdash qx[r]$ nebo zjednodušeně $pax \vdash qx$.

Definice 2.5. Nechť χ je konfigurace. M provede *nula přechodů* z χ do χ; zapisujeme:

$$\chi \vdash^0 \chi[\epsilon] \text{ nebo zjednodušeně } \chi \vdash^0 \chi.$$

Definice 2.6. Nechť $\chi_0, \chi_1, \ldots, \chi_n$ je sekvence přechodů konfigurací pro $n \geq 1$ a $\chi_{i-1} \vdash \chi_i[r_i]$ pro všechna $i = 1, \ldots, n$, což znamená:

$$\chi_0 \vdash \chi_1[r_1] \vdash \chi_2[r_2] \cdots \vdash \chi_n[r_n]$$

Pak M provede n přechodů z χ_0 do χ_n; zapisujeme:

$$\chi \vdash^n \chi[r_1 \ldots r_n] \text{ nebo zjednodušeně } \chi \vdash^n \chi.$$

Pokud $\chi_0 \vdash^n \chi_n[\rho]$ pro nějaké $n \geq 1$, pak

$$\chi_0 \vdash^+ \chi_n[\rho].$$

Pokud $\chi_0 \vdash^n \chi_n[\rho]$ pro nějaké $n \geq 0$, pak

$$\chi_0 \vdash^* \chi_n[\rho].$$

Teď již máme dostatečné teoretické znalosti, abychom mohli nadefinovat pojmy *přijímaný jazyk* a *ekvivalentní modely*.

Definice 2.7. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. *Jazyk přijímaný* konečným automatem M, $L(M)$, je definován:

$$L(M) = \{w : w \in \Sigma^*, sw \vdash^* f, f \in F\}.$$

Definice 2.8. Dva modely pro popis formálních jazyků (např. konečné automaty) jsou *ekvivalentní*, pokud specifikují tentýž jazyk.

2.2 Speciální typy

V následujících odstavcích budou popsány jednotlivé speciální typy konečných automatů. Každý typ bude formálně popsán, poté bude pospán algoritmus, jak převést libovolný konečný automat (splňující určité podmínky) na tento typ a nakonec budou uvedeny poznámky k implementaci daného algoritmu.

[1] pokud $a = \epsilon$, není ze vstupní pásky přečten symbol

Obecně k implementacím

K implementacím mohu již teď říct, že každý algoritmus je zapouzdřen do samostatné třídy. Tato třída přejímá rozhraní abstraktní třídy, která obsahuje následující virtuální metody:

- Provedení jednoho kroku převodu.
- Test na konec převodu.
- Řetězcové vyjádření stavu převodu.

Každá odvozená třída nadefinuje chování těchto tří metod.

Abstraktní třída obsahuje ještě jednu metodu, kterou je provedení celého převodu. Pozorného čtenáře již možná napadlo, že se bude jednat o cyklus (typu *while*), který v podmínce testuje konec převodu a v těle volá metodu pro provedení jednoho kroku převodu. Ještě pozornějšího čtenáře také možná napadlo, že tento mechanismus, kdy abstraktní třída definuje pořadí kroků algoritmu, ale jednotlivé kroky ponechá dodefinovat až své potomky, se podobá návrhovému vzoru *Šablonová metoda* [1, *Template Method*].

Řetězcové vyjádření umožňuje nahlédnout na aktuální stav převodu. Díky tomu můžeme provádět kroky převodu a průběžně si nechat vypisovat, co bylo provedeno. Aplikace toho využívá a tyto informace vypisuje, čímž se snaží přispívat k lepšímu pochopení konverzí konečných automatů.

Teoreticky by bylo možné, aby byl průběžně zobrazován i výstupní konečný automat ve své grafické reprezentaci. Takto to zatím implementováno nebylo, nicméně je to jedna z věcí, o které by mohla být rozšířena další verze tohoto programu.

Ještě je nutno podotknout, že třídy pracují s matematickým modelem konečného automatu, viz. 1.2.1.

2.2.1 Konečný automat bez ϵ-přechodů

Jak už bylo řečeno v úvodu této kapitoly, ϵ-přechody jsou v praxi nežádoucí. Jsou nežádoucí z tohoto důvodu, že simulace konečného automatu s ϵ-přechody může dospět např. ke konfiguraci, kdy bude mít na výběr mezi možností přečíst symbol ze vstupní pásky a přejít do stavu p nebo nečíst symbol ze vstupní pásky a přejít do stavu q. Otázka zní: *Kterou možnost vybrat?* Právě situace, kdy má algoritmus na výběr ze dvou (a více) možností, jsou nežádoucí. Vždy musí být na výběr maximálně jedna možnost. Tento problém je úzce spojen s problémem nedeterminismu, který je popsán v sekci 2.2.2.

Konečný automat bez ϵ-přechodů je definován následovně:

Definice 2.9. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. M je konečný automat bez ϵ-přechodů, pokud pro každé pravidlo $pa \rightarrow q \in R$, kde $p, q \in Q$, platí: $a \in \Sigma$ $(a \neq \epsilon)$.

Algoritmus

Pro odstranění ϵ-přechodů potřebujeme umět vypočítat tzv. *ϵ-uzávěr* pro každý stav. Neformálně můžeme říct, že ϵ-uzávěr stavu určuje všechny stavy, do kterých můžeme přejít bez přečtení vstupního symbolu. Formální definice vypadá takto:

Definice 2.10. Pro každý stav $p \in Q$ je definován *ϵ-uzávěr(p)*:

$$\epsilon\text{-uzávěr}(p) = \{q : q \in Q, p \vdash^* q\}$$

Při samotném převodu jsou nejprve do výstupního konečného automatu zkopírovány množina stavů Q a vstupní abecedu Σ. Stejně tak zůstává stejný počáteční stav s. Dál se vše řídí algoritmem 2.1.

Algoritmus 2.1: Odstranění ϵ-pravidel konečného automatu

Vstup: $M = (Q, \Sigma, R, s, F)$
Výstup: $M' = (Q, \Sigma, R', s, F')$ bez ϵ-přechodů

$R' = \emptyset$;
foreach $p \in Q$ **do**
    $R' = R' \cup \{pa \to q : p'a \to q \in R, a \in \Sigma, p' \in \epsilon\text{-}uzávěr(p), q \in Q\}$;
end
$F' = \{p : p \in Q, \epsilon\text{-}uzávěr(p) \cap F \neq \emptyset\}$;

Implementace

Třída, která implementuje převod, je v podstatě kopií uvedeného algoritmu. V konstruktoru třídy proběhne inicializace. Ze vstupního konečného automatu je zkopírována vstupní abeceda a název počátečního stavu (může být uložen již teď, protože během následujících operací se všechny stavy okopírují do výsledného konečného automatu). Výstupní konečný automat má zatím prázdnou množinu stavů a pravidel.

Pro indikaci konce převodu je vyhrazena členská proměnná ukazující na aktuálně zpracovávaný stav z množiny stavů vstupního konečného automatu. Převod je skončen, jakmile dosáhne konce této množiny.

Jeden krok převodu odpovídá jedné iteraci cyklu, tedy zpracování jednoho stavu podle výše uvedeného algoritmu. Během kroku se navíc aktuálně zpracovávaný stav přidá do množiny stavů výsledného konečného automatu.

Vkládání stavů je přesunuto až do tohoto místa z toho důvodu, že informace o tom, jestli je daný stav koncový, je součástí stavů a nikoli přímo součástí konečného automatu. Tím pádem musíme v době přidávání stavu vědět, jestli daný stav je nebo není koncový. A protože je pro zjištění této informace zapotřebí *ϵ-uzávěr(p)* stavu, počítáme koncovost (hlavně z důvodu optimalizace) v této fázi, kdy je již *ϵ-uzávěr(p)* znám.

2.2.2 Deterministický konečný automat

Nedeterminismus je v praxi nežádoucí stejně jako *ϵ-přechody*. Pokud je konečný automat nedeterministický, pak existuje konfigurace, v níž máme na výběr z více následujících konfigurací (tedy můžeme aplikovat jedno ze dvou a více pravidel).

Příklad 2.1. Příklad nedeterminismu z obrázku 1.2:

- aktuální konfigurace je $q1b$, nyní máme na výběr z pravidel $q1b \to q1$ a $q1b \to f$. Situace nám v tomto případě napoví, že aplikovat pravidlo, kdy můžeme přejít do koncového stavu a skončit tak čtení, je výhodnější, ale toto byl pouze lehký příklad pro ilustraci. Mohou se vyskytnou mnohem složitější případy, kdy výběr pravidla již není jednoduchý.

Deterministický konečný automat je definován následovně:

Definice 2.11. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat bez ϵ-přechodů. M je *deterministický konečný automat*, pokud pro každé $pa \to q \in R$ platí, že množina $R - \{pa \to q\}$ neobsahuje žádné pravidlo s levou stranou pa.

Algoritmus

Pro převod na deterministický konečný automat musí být vstupní konečný automat bez ϵ-přechodů. V knize A. Meduny [7] jsou uvedeny dva algoritmy, jak převést konečný automat na deterministický. První algoritmus generuje mnoho stavů, z nichž je dost nedostupných a musí proto následovat další fáze pro odstranění těchto stavů. Druhý tyto stavy negeneruje, a proto ho budeme používat. Pro pořádek si ještě nadefinujeme dostupný, resp. nedostupný stav:

Definice 2.12. Nechť $M = (Q, \Sigma, R, sF)$ je konečný automat. Stav $q \in Q$ je *dostupný*, pokud existuje $w \in \Sigma^*$, pro který platí $sw \vdash^* q$. Jinak q je *nedostupný*.

Každý stav nyní bude zastupovat množinu stavů. Nejprve je vytvořen nový počáteční stav, který reprezentuje množinu s počátečním stavem původního konečného automatu, a tento stav je vložen do množiny zpracovávaných stavů.

Každý stav z množiny zpracovávaných stavů je vyjmut, přidán do výsledného konečného automatu. Pro každý symbol abecedy je vytvořena množina stavů, do nichž existuje s daným symbolem pravidlo z některého ze stavů množiny, kterou zastupuje aktuálně zpracovávaný stav. Mezi aktuálním a vytvořeným stavem je do výstupního konečného automatu přidáno pravidlo s daným symbolem a vytvořený stav je přidán do množiny zpracovávaných stavů. Aktuálně zpracovávaný stav je koncový, pokud ve vstupním konečném automat byl koncový některý z množiny stavů, kterou zastupuje.

Celý postup je zobrazen na algortimu 2.2.

Algoritmus 2.2: Převod na deterministický konečný automat

Vstup: $M = (Q, \Sigma, R, s, F)$ bez ϵ-přechodů
Výstup: Deterministický konečný automat $M_d = (Q_d, \Sigma, R_d, s_d, F_d)$ bez nedostupných stavů

$s_d = \{s\}; Q_{new} = \{s_d\}; R_d = \emptyset; Q_d = \emptyset; F_d = \emptyset;$
repeat
  nechť $Q' \in Q_{new}; Q_{new} = Q_{new} - \{Q'\}; Q_d = Q_d \cup \{Q'\};$
  foreach $a \in \Sigma$ **do**
    $Q'' = \{q : p \in Q', pa \to q \in R\};$
    if $Q'' \neq \emptyset$ **then**
      $R_d = R_d \cup \{Q'a \to Q''\}$
    end
    if $Q'' \notin Q_d \cup \{\emptyset\}$ **then**
      $Q_{new} = Q_{new} \cup \{Q''\}$
    end
  end
  if $Q' \cap F \neq \emptyset$ **then**
    $F_d = F_d \cup \{Q'\};$
  end
until $Q_{new} = \emptyset$;

Implementace

V konstruktoru třídy je provedena inicializace, tak jak je uvedena v algoritmu. Je využito nového typu — množina množin stavů — z důvodu, kdy stav zastupuje množinu stavů.

Pro indikaci konce slouží fronta nových množin stavů (stavů, které mají být zpracovány). To odpovídá v algoritmu množině Q_{new} a převod tedy skončí v okamžiku, kdy je tato fronta prázdná.

2.2.3 Konečný automat bez neukončujících stavů

Neukončující stavy již nejsou pro praxi tak nevhodné jako *ϵ-přechody* nebo *nedeterminismus*. Z těchto stavů nelze nikdy přejít do koncového stavu a je jasné, že v nich nebude přijat žádný řetězec. Neukončující stav může mít v konečném automatu svůj význam, jak je uvedeno v kapitole 2.2.4. Obecně jsou ale neukončující stavy zbytečné a mohou být odstraněny bez jakýchkoli následků.

Definice 2.13. Nechť $M = (Q, \Sigma, R, s, F)$ je *deterministický konečný automat*. Stav $q \in Q$ je *ukončující*, pokud existuje řetězec $w \in \Sigma^*$, pro který platí $qw \vdash^* f, f \in F$. Jinak q je *neukončující*.

Algoritmus

Pro odstranění neukončujících stavů se očekává, že vstupní konečný automat bude deterministický.

Pokud se pokusíme neformálně popsat algoritmus 2.3, pak můžeme říct, že na začátku převodu přidáme do výsledného konečného automatu všechny koncové stavy, které zřejmě jsou ukončující. Poté postupně přidáváme stavy na levé straně pravidel, která vedou do některého z již přidaných stavů.

Algoritmus 2.3: Převod na konečný automat bez neukončujících stavů

Vstup: Deterministický konečný automat $M = (Q, \Sigma, R, s, F)$
Výstup: Deterministický konečný automat $M_t = (Q_t, \Sigma, R_t, s, F)$

$Q_0 = F; i = 0;$
repeat
  $i = i + 1;$
  $Q_i = Q_{i-1} \cup \{q : qa \to p \in R, a \in \Sigma, p \in Q_{i-1}\};$
until $Q_i == Q_{i-1}$;
$Q_t = Q_i;$
$R_t = \{qa \to p : qa \to p \in R, p, q \in Q_t, a \in \Sigma\}$

Implementace

Implementace se liší od algoritmu pouze v tom, že jakmile je nalezeno pravidlo, jehož stav by měl být přidán, je stav i pravidlo vloženo do výsledného konečného automatu. Nalezené pravidlo je uloženo do fronty nových pravidel, která slouží pro zjištění konce převodu. Jakmile v daném kroku nebyla nalezena žádná nová pravidla, převod je ukončen.

2.2.4 Dobře specifikovaný konečný automat

V praxi může být užitečné mít konečný automat, který se nikdy nezasekne. Tím je myšleno, že konečný automat dokáže v každé konfiguraci přejít do další konfigurace a přečíst tak celý řetězec. Takový konečný automat se nazývá úplný.

Definice 2.14. Nechť $M = (Q, \Sigma, R, s, F)$ je *deterministický konečný automat*. M je *úplný*, pokud pro libovolné $p \in Q$, $a \in \Sigma$ existuje právě jedno pravidlo $pa \to q \in R$ pro nějaké $q \in Q$. Jinak M je *neúplný*.

Dobře specifikovaný konečný automat využívá ve své definici úplný konečný automat.

Definice 2.15. Nechť $M = (Q, \Sigma, R, s, F)$ je *úplný konečný automat*. Pak M je *dobře specifikovaný konečný automat*, pokud:

1. Q nemá nedostupné stavy,
2. Q má maximálně jeden neukončující stav[2].

K tomu, abychom získali dobře specifikovaný konečný automat, musíme převést konečný automat na úplný. Tento převod popisuje algoritmus 2.4. Podmínkou převodu je, že vstupní konečný automat musí být deterministický.

Algoritmus

Převod na úplný konečný automat je velmi jednoduchý. Do konečného automatu je přidán stav simulující „past“ a do tohoto stavu jsou svedena pravidla pro všechny stavy a všechny symboly abecedy, pro něž ještě pravidla neexistují. Algoritmus 2.4 zobrazuje celý postup.

Algoritmus 2.4: Převod na úplný konečný automat

Vstup: Neúplný deterministický konečný automat $M = (Q, \Sigma, R, s, F)$
Výstup: Úplný deterministický konečný automat $M_c = (Q_c, \Sigma, R_c, s, F)$

$Q_c = Q \cup \{q_{false}\}$;
$R_c = R \cup \{qa \to q_{false} : a \in \Sigma, q \in Q_c, qa \to p \notin R, p \in Q\}$

Implementace

Při inicializace je do výsledného konečného automatu kompletně zkopírován vstupní konečný automat. Navíc je vložen nový stav „past“.

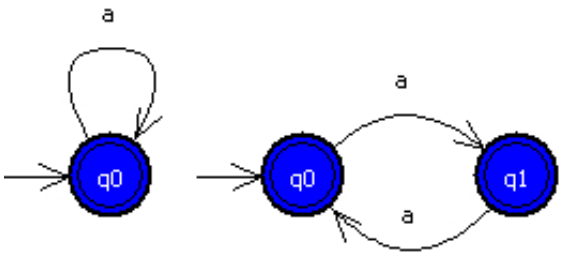
Pro všechny stavy výstupního konečného automatu jsou poté vyhledány symboly, pro něž z daného stave neexistuje pravidlo, a tyto jsou svedeny do „pasti“.

Jelikož se algoritmus týká pouze stavů, převod je ukončen v okamžiku, kdy byla výše uvedená operace provedena pro všechny stavy.

[2]Pokud dobře specifikovaný konečný automat má neukončující stav, je to q_{false} z algoritmu pro úplný konečný automat.

2.2.5 Minimální konečný automat

Jeden jazyk může být vyjádřen mnoha *ekvivalentními modely*. Na obrázku 2.1 jsou dva konečné automaty. Oba specifikují stejný jazyk. Z těchto dvou je ale jeden minimální. Naopak ten druhý obsahuje nadbytečný stav a pravidlo. Pokud tedy chceme dostat minimální řešení, pak musíme konečný automat minimalizovat.



Obrázek 2.1: Ekvivalentní konečné automaty

Abychom mohli nadefinovat *minimální konečný automat*, potřebujeme znát pojem *rozlišitelné stavy*.

Definice 2.16. Nechť $M = (Q, \Sigma, R, s, F)$ je dobře specifikovaný konečný automat a nechť $p, q \in Q, p \neq q$. Stavy p a q jsou *rozlišitelné*, pokud existuje řetězec $w \in \Sigma^*$ takový, že: $pw \vdash^* p'$ a $qw \vdash^* q'$, kde $p', q' \in Q$ a $((p' \in F$ a $q' \notin F)$ nebo $(p' \notin F$ a $q' \in F))$. Jinak stavy p a q jsou *nerozlišitelné*.

Množina stavů minimálního konečného automatu obsahuje pouze rozlišitelně stavy. Formální definice tedy zní:

Definice 2.17. Nechť $M = (Q, \Sigma, R, s, F)$ je dobře specifikovaný konečný automat. Potom M je *minimální konečný automat*, pokud M obsahuje pouze rozlišitelné stavy.

Algoritmus

V algoritmu 2.5 je znázorněn postup převodu na minimální konečný automat. Nejprve jsou vytvořeny dva stavy — stav zastupující množinu koncových stavů a stav zastupující množinu nekoncových stavů. Pokud stav (množina stavů) splňuje podmínku, že některé jeho stavy z množiny, kterou reprezentuje, přechází do jiného stavu (množiny stavů) než ostatní stavy z množiny, kterou reprezentuje, je tento stav (množina stavů) rozštěpen.

Implementace

Při inicializaci je zkopírována abeceda konečného automatu a jsou vytvořeny dvě množiny stavů — jedna obsahující koncové stavy a druhá obsahující nekoncové stavy. Obě množiny jsou vloženy do množiny dočasných stavů, které zastupují množiny stavů. Stavy nejsou vkládány do výsledného konečného automatu, dokud není skončeno štěpení.

Převod se skládá z dvou fází. V první fázi probíhá štěpení stavů podle cyklu uvedeného v algoritmu. Jedna iterace cyklu odpovídá rozštěpení jednoho stavu. Pro indikaci konce cyklu tedy slouží příznak, jestli bylo při poslední iteraci provedeno štěpení. Po poslední

Algoritmus 2.5: Převod na minimální konečný automat

Vstup: Dobře specifikovaný konečný automat $M = (Q, \Sigma, R, s, F)$
Výstup: Minimální konečný automat $M_m = (Q_m, \Sigma, R_m, s_m, F_m)$

$Q_m = \{\{p : p \in F\}, \{q : q \in Q - F\}\}$;
repeat
  if *existuje* $X \in Q_m, d \in \Sigma, X_1, X_2 \subset X$ *takové, že:* $X = X_1 \cup X_2, X_1 \cap X_2 = \emptyset$ **and** $\{q_1 : p_1 \in X_1, p_1 d \to q_1 \in R\} \subseteq Q_1, Q_1 \in Q_m, \{q_2 : p_2 \in X_2, p_2 d \to q_2 \in R\} \cap Q_1 = \emptyset$ **then**
    rozštěp X na X_1 a X_2 v Q_m
  end
until *není možné provést žádné další štěpení* ;
$R_m = \{Xa \to Y : X, Y \in Q_m, pa \to q \in R, p \in X, q \in Y, a \in \Sigma\}$;
$s_m = X : s \in X$;
$F_m = \{X : X \in Q_m, X \cap F \neq \emptyset\}$;

iteraci cyklu jsou stavy vloženy do výsledného konečného automatu a současně jsou nastavovány koncové stavy, protože tuto informaci je nutné znát již při vkládání stavů. V tento okamžik se přechází do druhé fáze.

Ve druhé fázi jsou vytvářena pravidla, jak je uvedeno v algoritmu. Jakmile jsou projita všechna pravidla vstupního konečného automatu, skončí druhá fáze a tak i celý převod.

Kapitola 3

Ilustrace činnosti

Při ilustraci činnosti lze ověřovat, jestli konečný automat přijímá nebo nepřijímá určitý řetězec. Toho lze využít např. pro ověřování přijímaného jazyka nebo pro zjišťování, jak konečný automat funguje.

3.1 Teorie ilustrace činnosti

Ilustrace činnosti konečného automatu $M = (Q, \Sigma, R, s, F)$ spočívá v tom, že je na vstupní pásku vložen řetězec $w \in \Sigma^*$ a z počáteční konfigurace sw jsou prováděny přechody.

Během přechodů mohou nastat tyto dvě situace:

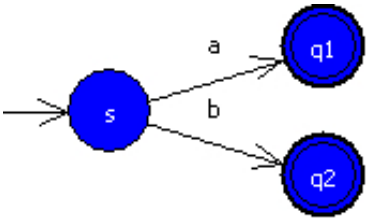
1. Celý řetězec je přečten, konečný automat skončil ve stavu q. Pak:
   (a) pokud $q \in F$, konečný automat M přijímá tento řetězec, tedy $w \in L(M)$, kde $L(M)$ značí jazyk přijímaný konečným automatem M.
   (b) pokud $q \notin F$, řetězec není přijat a platí $w \notin L(M)$.
2. Neexistuje následující konfigurace (řetězec nebyl přečten celý), proto $w \notin L(M)$.

Řetězec je přijat pouze v případě, že je přečten celý a čtení skončilo v koncovém stavu. Pokud se konečný automat dostane do konfigurace, ve které již nelze přejít do žádné další z důvodu, že nebylo nalezeno pravidlo, zasekne se. To mimo jiné také znamená to, že řetězec není součástí jazyka (tedy $w \notin L(M)$).

Situace, kdy není možné přejít do další konfigurace a řetězec není celý přečten, se dá řešit jinak. Alespoň v praxi, kde bývají konečné automaty navrhovány pro lexikální analýzu, toho bývá často využíváno. Řetězec v tomto případě není ihned odmítnut, ale zjišťuje se, jestli stav, ve kterém nebyla nalezena další konfigurace, není koncový. Pokud ano, pak se přijme pouze část tohoto řetězce a čtení pokračuje z počátečního stavu se zbytkem řetězce.

Formálněji zapsáno může vše vypadat asi takto: Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. Mějme $u, v \in \Sigma^*, a \in \Sigma$ takové, že $uav = w$. Pokud jsme v konfiguraci qav takové, že $sw \vdash^* qav$, kde $q \in F$ a současně $qa \to p \notin R$ pro libovolné $p \in Q$, pak $u \in L(M)$ a pokračujeme z konfigurace sav.

Příklad 3.1. Konečný automat na obrázku 3.1 (formálně definován jako $M = (Q = \{s, q1, q2\}, \Sigma = \{a, b\}, R = \{sa \to q1, sb \to q2\}, s, F = \{q1, q2\})$) má na vstupní pásce řetězec $w = ab$. Posloupnost konfigurací by byla: $sab \vdash q1b[sa \to q1]$. Za normálních okolností by nebylo možné tento řetězec přijmout, protože neexistuje žádná další konfigurace,



Obrázek 3.1: Příklad konečného automatu

přitom řetězec ještě není přečten celý. Pokud budeme vycházet z předchozího odstavce, pak v konfiguraci $q1b$ neexistuje žádné $q1b \rightarrow p \in R$ pro libovolné $p \in Q$, ale současně $q1 \in F$. Pak $a \in L(M)$ a čtení bude pokračovat z konfigurace sb.

V praxi také může dojít k situaci, že v dané konfiguraci není nalezeno pravidlo (tedy není žádná další konfigurace), ale také stav, ve kterém jsme skončili, není koncový. Pokud chceme pokračovat ve čtení, nabízí se řešení, že řetězec, který jsme načetli, zahodíme a provádíme přechody znovu z počátečního stavu se zbytkem řetězce.

Formální popis této situace by byl podobný jako v případě, kdy jsme byli v koncovém stavu, tedy: Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. Mějme $u, v \in \Sigma^*, a \in \Sigma$ takové, že $uav = w$. Pokud jsme v konfiguraci qav takové, že $sw \vdash^* qav$, kde $q \in Q, q \notin F$ a současně $qa \rightarrow p \notin R$ pro libovolné $p \in Q$, pak $u \notin L(M)$ a pokračujeme z konfigurace sav.

Může docházet ještě ke dvěma komplikacím:

1. Hned v počáteční konfiguraci neexistuje žádná následující konfigurace, a proto opakovaně dochází k přecházení do počátečního stavu a opakovaně je přijímán nebo zahazován prázdný řetězec podle toho, jestli je nebo není počáteční stav zároveň koncový.

2. Konečný automat obsahuje stav simulující „past“. Tento stav se stará o to, aby pro každý stav a každý symbol existovalo pravidlo, čili v konečném automatu bude vždy existovat následující konfigurace.

U obou problémů již záleží na programátorovi, jak se s nimi vypořádá. Mé řešení bude popsáno v následující části.

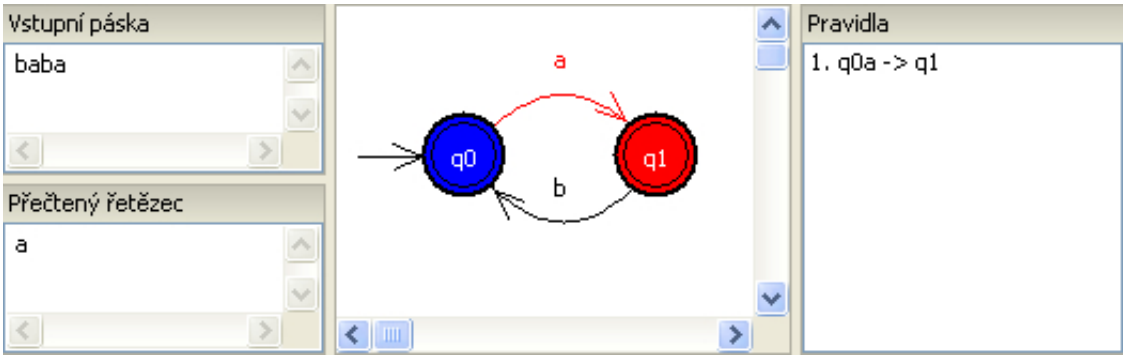
Konečný automat, jehož ilustraci chceme provést, musí být deterministický. Pokud by obsahoval ϵ-pravidla nebo nedeterminismus, pak může docházet k nejednoznačnostem, jak je popsáno v sekcích 2.2.1 a 2.2.2.

3.2 Implementace

3.2.1 Zobrazení

Implementace ilustrace činnosti pracuje s grafickým modelem konečného automatu. Konečný automat je vykreslován do speciálního okna. Navíc je vždy barevně odlišen stav aktuální konfigurace a pravidlo, které bylo pro tuto konfiguraci aplikováno. Na začátku se jedná o počáteční stav a šipku označující počáteční stav.

V okně, kde probíhá ilustrace, jsou navíc další tři podokna. První, do něhož je možné psát text, zastupuje vstupní pásku. Druhé vypisuje již přečtený řetězec a třetí zobrazuje pravidla, která byla aplikována.



Obrázek 3.2: Ukázka programové ilustrace konečného automatu

3.2.2 Módy ilustrace

Ilustrace může probíhat ve dvou módech. V prvním módu probíhá čtení jako při lexikální analýze (rozděluje vstupní text na lexémy). Vždy je přečten celý řetězec. Z počáteční konfigurace konečného automatu jsou prováděny přechody až do okamžiku, kdy neexistuje žádná další konfigurace. Pak pokud je stav aktuální konfigurace koncový, načtená část řetězce je přijata a přechází se opět do počátečního stavu, odkud jsou prováděny přechody se zbylou částí řetězce. Pokud stav aktuální konfigurace není koncový, načtený řetězec je přeskočen a prohlášen za chybný. Pokračuje se opět z počátečního stavu se zbylým řetězcem.

Situace, kdy ihned v počátečním stavu neexistuje následující konfigurace z důvodu, že nebylo nalezeno pravidlo, je u ilustrace ohlídána také. Pokud je počáteční stav zároveň koncový, docházelo by k opakovanému přijímání prázdného řetězce. Ošetřeno je to tak, že prázdný řetězec je přijat pouze v případě, že vstupní páska je již prázdná. Jinak je přeskočen jeden symbol na vstupu. Jestliže počáteční stav není koncový, je jeden symbol na vstupu přeskočen také.

V tomto módu je také ošetřena situace, kdy konečný automat obsahuje stav „past“. Algoritmus pro provedení přechodu si hlídá, jestli nevstupuje do takového stavu. Jakmile by do něj vkročil, celý řetězec by byl přečten a ve finále nepřijat (což je správná funkce tohoto stavu). Proto se při každém přechodu zjišťuje, jestli se nechystáme udělat krok do „pasti“, a v případě že ano, tento přechod není uskutečněn a algoritmus se zachová stejně, jako by neexistovala následující konfigurace.

Pro úplnost je zde uvedena definice stavu „past“, jak je používána v programu.

Definice 3.1. Nechť $M = (Q, \Sigma, R, s, F)$ je konečný automat. Stav q je stav typu „past“, pokud platí $q \notin F$ a $qa \rightarrow q \in R, \forall a \in \Sigma$.

Ve druhém módu není přecházeno do počátečního stavu a v případě, že v nějaké konfiguraci neexistuje žádná další konfigurace, řetězec není přijat. Řetězec je přijat pouze v případě, že je přečten celý a je přijat v koncovém stavu.

3.2.3 Řízení ilustrace

Ilustraci je možné řídit pomocí kroků vpřed a vzad. Při provedení kroku vpřed je zavolán algoritmus 3.1. V algoritmu uvedené akce *přidej a proved'* způsobí vytvoření a provedení příkazu. Příkaz (implementován jako návrhový vzor *Příkaz* [1, *Command*]) je objektem,

který provede určitou akci. Tuto akci však umí také vrátit zpět (odvolat), čehož je využito při provedení kroku vzad.

Implementovány jsou tři typy příkazů:

1. *Příkaz pro přechod* je složen z těchto akcí:
   - odebrání prvního symbolu ze vstupní pásky,
   - přidání symbolu do přečteného řetězce,
   - nastavení nového aktuálního pravidla a stavu.
2. *Příkaz pro přijetí* řetězce provádí dvě akce:
   - vyprázdnění přijatého řetězce,
   - přechod do počátečního stavu.
3. *Příkaz pro přeskočení* provádí tři akce:
   - odebrání prvního symbolu ze vstupní pásky (tato akce je pouze u přeskakovacího příkazu 1),
   - vyprázdnění přijatého řetězce,
   - přechod do počátečního stavu.

Algoritmus 3.1: Algoritmus pro provedení kroku vpřed

```
if nalezen následující stav then
    if první mód and následující stav je past then
        if aktuální stav je koncový then přidej a proveď restartovací příkaz;
        else řetězec nepřijat;
    else
        přidej a proveď přechodový příkaz;
    end
else
    if vstupní páska prázdná then
        if aktuální stav je koncový then řetězec přijat;
        else řetězec nepřijat;
    else
        if první mód then
            if přečten prázdný řetězec then
                přidej a proveď přeskakovací příkaz 1;
            else
                if aktuální stav je koncový then přidej a proveď restartovací příkaz;
                else přidej a proveď přeskakovací příkaz 2;
            end
        else
            řetězec nepřijat;
        end
    end
end
```

Kapitola 4

Export

Pod pojmem export je myšleno vyjádření konečného automatu jinou formou. Tou je v aplikaci zdrojový kód v jazyce C nebo obrázek.

4.1 Jazyk C

Při exportu do jazyka C vznikne zdrojový kód, který po přeložení a spuštění odsimuluje činnost konečného automatu, jak je popsána v kapitole 3.2.2 (v prvním módu). Konečný automat se tedy chová jako lexikální analyzátor.

Vytvořený zdrojový kód je složen ze dvou částí. První částí je pevně daná kostra zdrojového souboru, která obsahuje většinu funkcí pro vytvoření lexikálního analyzátoru. Tato kostra je uložena v souboru. Druhou částí je kód, který se liší s každým různým konečným automatem. Tato část je programově generována a vložena do výsledného zdrojového souboru.

Kostra zdrojového souboru se snaží podobat zdrojovému souboru, který bychom vygenerovali pomocí nástroje *Flex* [2]. Flex je určen ke generování programů pro zpracování textových souborů a bývá využíván právě pro vytváření lexikálních analyzátorů. Názvy důležitých datových typů, konstant, proměnných a funkcí budou vyjmenovány v části 4.1.2.

4.1.1 Generování kódu

Generování kódu pracuje s matematickým modelem konečného automatu, grafické informace nejsou vůbec potřeba. Exportovaný konečný automat musí být nejméně deterministický. Pokud by obsahoval ϵ-pravidla nebo nedeterminismus, pak může docházet k nejednoznačnostem, jak je popsáno v sekcích 2.2.1 a 2.2.2.

Třída pro export je implementována jako vzor *Návštěvník* [1, *Visitor*]. Třída konečného automatu obsahuje metodu, která přijme jako návštěvníka objekt exportu, a tato metoda zavolá přijmutí také nad svými stavy a pravidly. „Přijímací metoda“ konečného automatu, stavů a pravidel volá metodu objektu návštěvníka a umožní mu tak získat přehled o struktuře konečného automatu.

U objektu konečného automatu si uloží název počátečního stavu a vstupní abecedu. Pro ukládání stavů a pravidel v návštěvníkovi slouží speciální struktura, která mapuje stavy na seznam pravidel. Jedná se tedy o pole, kde indexem je stav a hodnotou je seznam pravidel. Při návštěvě stavu je na indexu s názvem stavu vytvořen prázdný seznam pravidel (pokud ještě neexistuje — existovat by mohl, kdyby konečný automat volal přijmutí návštěvníka

pravidly dřív než stavy) a při návštěvě pravidla je na index se zdrojovým stavem (stavem na levé straně pravidla) přidáno pravidlo na konec seznamu.

Při ukládání jsou současně upravovány názvy stavů a to jak ve stavech samotných, tak i u pravidel. Název stavu je převeden, jak bylo uvedeno v části 1.2.1. Znaky {}, se nahradí znakem podtržítka a malé znaky jsou převedeny na velké. Omezení názvů stavů bude zřejmé za chvíli — název bude tvořit identifikátor, pro který v jazyce C platí jistá omezení.

Díky výše uvedenému způsobu uložení máme usnadněno vytváření programově generované části, zvláště pak poslední bod uvedený v následujícím výčtu. Programově generovaná část zahrnuje:

- deklaraci výčtového typu stavů,
- konstantu s identifikátorem počátečního stavu,
- definici pole koncových stavů a jeho délku,
- konstantu s identifikátorem stavu „past“,
- definici funkce pro hledání následujícího stavu.

Ve výčtovém typu stavů jsou jako symbolické názvy konstant uváděny názvy stavů (již upravené). Výčtový typ navíc obsahuje dva speciální identifikátory pro:

- žádný stav,
- neplatný stav.

Jejich význam bude vysvětlen později.

Konstanta s identifikátorem počátečního stavu obsahuje identifikátor, který má ve výčtovém typu stavů počáteční stav konečného automatu.

Pole koncových stavů je polem konstant výčtového typu stavů, které jsou koncové.

Konstanta s identifikátorem stavu „past“ zastupuje identifikátor stavu, který se chová jako past. Je důležité tento stav znát, abychom mohli zabránit přechodu do něj. Pokud konečný automat nemá žádný stav typu „past“, je hodnota této konstanty taková, aby neodpovídala žádné z výčtového typu stavů.

Funkce pro hledání následujícího stavu je přepínač, který přesně odpovídá pravidlům konečného automatu. Část funkce je zachycena v algoritmu 4.1. Poprvé je využito speciálního stavu *žádný stav*, která je vrácena, pokud nebyl nalezen následující stav.

Při vytváření této funkce je procházena mapa stavů na seznam pravidel. Pro každý stav je vytvořena **case** větev a pro každé pravidlo ze seznamu daného stavu je vytvořena podmínka **if** se symbolem daného pravidla. Návratovou hodnotou v této podmínce je identifikátor cílového stavu (stavu na pravé straně daného pravidla). Pokud je v jednom seznamu více pravidel se stejným cílovým stavem a pokud jsou symboly těchto pravidel v řadě za sebou, jsou tyto symboly shlukovány a místo toho, aby byla vytvořena podmínka **if** pro každý symbol zvlášť, je vytvořena jedna podmínka **if** pro interval symbolů.

Příklad 4.1. Množina pravidel je $R = \{q1a \rightarrow q2, q1b \rightarrow q2, q1c \rightarrow q2\}$). Vygenerovaný zdrojový kód pro stav $q1$ by vypadal následovně. Protože všechna pravidla z tohoto stavu mají stejný cílový stav, tak namísto toho, aby byly vytvořeny tři podmínky **if**(*symbol == 'a'*) ..., **if**(*symbol == 'b'*) ..., **if**(*symbol == 'c'*) ..., je vytvořena jedna podmínka **if**(*symbol >= 'a'* **and** *symbol <= 'c'*) ...

Algoritmus 4.1: Hledání následujícího stavu na základě aktuálního stavu a znaku

```
switch aktuální stav do
    case identifikátor stavu p
        if aktuální znak == a then
            // pravidlo pa → q
            return identifikátor stavu q;
        else
            // žádné pravidlo s tímto symbolem
            return žádný stav;
        end
    end
    // pro všechny stavy a pravidla
    ...
end
```

4.1.2 Popis kódu

V této části budou popsány důležité datové typy, konstanty, proměnné a funkce kostry zdrojového kódu konečného automatu.

Datové typy

- yy_char — datový typ znaku.
- yy_int — datový typ celého čísla (*int*).
- yy_size_t — datový typ pro celá kladná čísla.
- yy_bool — datový typ pro boolean hodnotu.
- YY_BUFFER_STATE — struktura s informacemi o bufferu.

Konstanty

- YY_NULL — návratová hodnota funkce *yyinput* při dosažení EOF.
- YY_BUFFER_SIZE — velikost bufferu, do nějž se načítá vstupní text.
- YY_TOKEN_SIZE — výchozí délka textu tokenu.

Proměnné

- yyin — vstupní soubor typu FILE*; pokud není uveden, je použit *stdin*.
- yyout — výstupní soubor typu FILE*; pokud není uveden, je použit *stdout*.
- yy_current_buffer — proměnná typu YY_BUFFER_STATE, která nese informace o bufferu.
- yyval — hodnota tokenu, obsahuje jeden z výčtového typu stavů.
- yytext — text tokenu; je automaticky vytvořen, ale musí být explicitně smazán funkcí *yy_delete_token*.
- yyleng — délka textu tokenu.

- yysize — velikost paměti tokenu (skutečný počet alokovaných znaků).

Funkce

- yyinput — načtení aktuálního znaku na vstupu.
- yyunput — vrácení znaku zpět na vstup.
- yy_init_buffer — inicializace bufferu (voláno automaticky při prvním zavolání funkce *yylex*).
- yy_get_next_buffer — načtení dalšího textu do bufferu ze vstupu (voláno automaticky funkcí *yyinput*).
- yy_create_token — vytvoření tokenu (voláno automaticky při prvním zavolání funkce *yylex*).
- yy_realloc_token — rozšíření velikosti pole tokenu (voláno automaticky funkcí *yy_put_to_token*).
- yy_delete_token — uvolnění paměti tokenu (nutné volat explicitně na konci programu).
- yy_reset_token — vymazání obsahu textu tokenu.
- yy_put_to_token — přidání znaku na konec textu tokenu.
- yy_wlex_error — výpis chybové hlášky.
- yy_fatal_error — výpis chybové hlášky a konec programu.
- yylex — nejdůležitější funkce, její popis je uveden dále.

Funkce *yylex* rozděluje vstupní text do lexikálních jednotek (lexémů), které jsou reprezentovány *tokeny*. Návratovou hodnotou funkce je jedna z hodnot výčtového typu stavů. Pokud funkce vrátí hodnotu *žádného stavu*, pak bylo dosaženo konce vstupního souboru. Hodnota *neplatný stav* je vrácena v případě, že některá část vstupního řetězce není přijímána konečným automatem. V případě, že některý řetězec je přijat, je návratovou hodnotou identifikátor stavu, v němž byl řetězec přijat. Navrácená hodnota je současně uložena do proměnné *yyval* a načtený text do *yytext*.

4.1.3 Překlad kódu

Kód je generován podle normy C99 jazyka C [4], proto by podle toho měl být nastaven i překlad. Program umí vygenerovat i překladový soubor — *Makefile*. Postup generování je obdobný jako při generování zdrojového kódu — základem je kostra a doplní se některé specifické údaje, v tomto případě pouze název zdrojového souboru pro překlad.

4.2 Obrázek

Druhou formou exportování je obrázek. Tato operace pracuje samozřejmě s grafickým modelem konečného automatu. Jedná se o jednoduchou akci, při níž se všechny stavy a pravidla vykreslují do pomocného bufferu. Současně je počítána oblast, kterou konečný automat (tedy stavy a pravidla) zabírá, aby výsledný obrázek měl co nejmenší rozměry a aby neobsahoval velkou volnou plochu.

Vytvořený buffer má rozměry celého konečného automatu. Jakmile je vykreslení dokončeno, převádí se pouze spočítaná oblast do bitové mapy, z níž je následně vygenerován výstupní obrázek ve formátu BMP nebo PNG.

Kapitola 5

Ostatní

V této kapitole budou popsány některé důležitější vlastnosti aplikace, které nemohly být zařazeny do žádné z předchozích kapitol a také jejich rozsah nevydá za celou kapitolu.

5.1 XML

Formát XML je používán pro ukládání/načítání konečných automatů do/ze souboru. Pro práci s těmito soubory je použita knihovna TinyXml [9]. Ukládán je grafický model konečných automatů, protože požadujeme přesné zachování jeho vzhledu i po znovunačtení souboru. Pokud bychom ukládali matematický model, informace o grafickém rozložení by byly ztraceny.

Soubor obsahuje následující data:

- informace o konečném automatu:
  - rozměry,
  - abecedu,
  - výchozí nastavení stavů (barvy, písmo),
  - výchozí nastavení pravidel (písmo),
  - barvu písma a pozadí,
- informace o stavech, u každého stavu pak:
  - název,
  - příznak, jestli je koncový,
  - pozici stavu,
- informace o pravidlech, u každého pravidla:
  - zdrojový stav,
  - cílový stav,
  - čtený symbol,
  - řídící body křivky,
  - pozici zobrazovaných symbolů.,
- informace o počátečním stavu:
  - zdrojový bod,
  - cílový stav.

Ukládání a načítání zajišťuje třída, která odpovídá návrhovému vzoru *Návštěvník* [1, *Visitor*].

5.1.1 Ukládání

Konečný automat přijme objekt třídy pro ukládání, čímž mu umožní vytvořit si strukturu XML elementů popisující daný konečný automat podle výše uvedeného schématu. Tato

struktura je uložena do objektu TinyXml dokumentu, který je pak již pouze zapsán do souboru na disk.

5.1.2 Načítání

Při načítání je postup obrácený než u ukládání. To znamená, že nejprve je z disku načten soubor do objektu TinyXml dokumentu. Objekt provede základní analýzu souboru. Zjistí tedy, jestli je soubor syntakticky správný podle pravidel XML. Třída pro načítání následně prochází strukturu TinyXml dokumentu a přitom vytváří jednotlivé objekty (nejprve konečný automat samotný, poté stavy a na konec pravidla). Pořadí elementů je přesně dáno, a proto jakmile některý očekávaný element chybí, načítání skončí s chybou.

5.2 Jazyk

Všechny řetězcové konstanty použité v programu jsou uloženy v souboru na disku. Z tohoto souboru jsou při startu načteny a použity v programu. Díky tomu je možné překládat aplikaci do několika jazyků. Stačí vytvořit nový jazykový soubor (nejlépe jako kopii některého existujícího jazykového souboru) a zkopírovat ho do složky *language* v kořenové složce programu. V programu lze pak změnit aktuální jazyk v nastavení aplikace. Pro úplný překlad je také nutné přeložit nápovědu k programu, jejíž soubory se nachází ve složce *help* v kořenové složce programu, a text dialogu „*O programu*“ v téže složce.

Třída, která se stará o načítání textů daného jazyka, je implementována jako návrhový vzor *Jedináček* [1, *Singleton*], čili má jedinou instanci. Při startu je nutné volat statickou metodu třídy pro inicializaci, které je předán název souboru s řetězci. Pokud by třída nebyla správně inicializována, bude při žádosti o některý text vrácen jen prázdný řetězec.

Pokud v programu požadujeme některý text, je volána opět statická metoda, v níž je požadovaný řetězec identifikován na základě celočíselné konstanty, kterou má uvedenou v souboru. Tyto celočíselné konstanty jsou pevně dány a nesmí být změněny. Ve zdrojovém kódu třídy jsou tyto konstanty uvedeny v rámci výčtového typu a mají přidělena symbolická jména. Jméno je uvedeno i v souboru s řetězci, ale v programu není nijak použito.

Příklad 5.1. Příklad souboru s jedním záznamem:

```
<?xml version='1.0' encoding='utf8'?>
<language type='czech' short='cs' name='Česky'>
   <item id='1' name='SYMBOLICKY_NAZEV'>
      <text>Text</text>
   </item>
</language>
```

Kořenový prvek obsahuje tři atributy, kde prvním je anglický název jazyka, druhým je zkratka jazyka a třetím je originální název jazyka. Anglický název není v programu zatím použit. Zkratka jazyka je naproti tomu velmi důležitá. Ve složce s názvem tvořeným touto zkratkou musí být umístěna nápověda k programu. Zde ji totiž bude program hledat, pokud bude tento jazyk zvolen. Zkratka jazyka je také součástí názvu souboru s textem dialogu „*O programu*“. Třetí atribut bude zobrazen v nastavení aplikace u volby jazyka.

Záznam má pak dva atributy, z nichž první určuje identifikátor řetězce a druhým je symbolické jméno, jak je uvedeno ve zdrojovém souboru třídy pro načítání řetězců jazyka. Záznam také obsahuje jeden element, v němž je uveden samotný řetězec v daném jazyce.

5.3 Rozmístění stavů

Na problém rozmístění stavů jsme narazili již v kapitole Modely konečného automatu (1.2), kde bylo řečeno, že pokud se snažíme převést matematický model na grafický, chybí nám informace o umístění stavů nebo o řídících bodech pravidel. Převod z matematického na grafický model je poměrně častou operací. Pokaždé, když chceme provést některý algoritmus z kapitoly Převody (2), proběhnou tyto operace:

1. Konverze grafického modelu na matematický
2. Aplikace algoritmu
3. Konverze matematického modelu na grafický

Během aplikace algoritmu dochází k přidávání, spojování a mazání stavů a přidávání a mazání pravidel. Nelze tedy ani použít původní souřadnice či řídící body.

V mé práci se mi bohužel nepodařilo implementovat algoritmus, který by dokázal rozmístit stavy a pravidla inteligentně, tedy tak, aby stavy neležely přes sebe a křivky pravidel se nekřižovaly. Hledal jsem inspiraci u software *Graphviz* (konkrétně práce pojednávající o tématu kreslení grafů [3]), který je určen pro zobrazování grafů, ale zjistil jsem, že implementovat algoritmus, který by dokázal provést inteligentní rozmístění, by vydalo na jednu bakalářskou práci a na to již bohužel nezbyl čas. Rozhodně je to jedna z částí, která by mohla být upravena v další verzi programu.

Rozmístění stavů po konverzi z matematického na grafický model samozřejmě proběhne. Systém rozmístění je však velmi jednoduchý. Jedná se o třídu, která umísťuje stavy do mřížky. Začíná v levém horním rohu plochy grafického modelu konečného automatu a pokračuje po řádcích vpravo a dolů. Počáteční stav je umístěn vždy na první pozici, tedy v levém horním rohu.

Tento způsob rozmístění vyřeší problém s překrývajícími se stavy, ale křivky pravidel se kříží stále.

5.4 Příkazy konečného automatu

V kapitole o grafickém modelu konečného automatu (1.2.2) se objevila zmínka o příkazech konečného automatu jako o akcích, které je možné provést a případně odvolat, vzít zpět. Tyto příkazy jsou implementovány jako návrhový vzor *Příkaz* [1, *Command*]. Každá akce (příkaz) implementuje rozhraní abstraktní třídy příkazu, která obsahuje dvě důležité metody:

- proveď příkaz a
- odvolej příkaz.

Příkazy konečného automatu jsou téměř všechny akce, které jsou nad ním prováděny, např. přidání stavu, přidání pravidla, smazání stavu, smazání pravidla, nastavení počátečního stavu, nastavení abecedy, přesun stavu, atd.

Objekt konečného automatu si tyto příkazy uchovává v dynamickém poli. Na aktuální příkaz ukazuje index. Konečný automat dále obsahuje metody pro přidání příkazu, pro odvolání příkazu a pro znovuprovedení příkazu. Metoda pro přidání příkazu automaticky zavolá metodu pro provedení příkazu. Jakmile je voláno odvolání příkazu, je nad aktuálním příkazem volána metoda pro odvolání a posune se index o jeden příkaz zpět. Příkaz však

ještě není smazán z pole, aby mohla být volána metoda pro znovuprovedení příkazu. Až když je přidáván nový příkaz, jsou všechny příkazy od aktuálního až na konec smazány, nový příkaz je vložen za aktuální a index aktuálního je posunut.

5.5 Překlad projektu

Pro překlad projektu je nutné mít nainstalovaný překladač *GNU C++* a knihovnu *wxWidgets* ve verzi alespoň 2.8. Projekt je přenositelný mezi platformami *Windows* a *Linux*. Na obou by měl běžet bez problémů.

5.5.1 Linux

Překladač lze lehce nainstalovat pomocí balíčků. U wxWidgets ale doporučuji vlastní překlad a instalaci. Verzí zdrojových kódů *wxWidgets* pro Linux je archiv *wxGTK*, který lze stáhnout ze stránek www.wxwidgets.org. Po stáhnutí a rozbalení archivu se přepneme do složky `build`, v níž se nachází skript `configure`. Pro úspěšný překlad projektu je nutné mít knihovnu *wxWidgets* přeloženou v tzv. *Unicode build* a ve verzi *Release*. Aby bylo tohoto docíleno, musí překlad knihovny vypadat takto:

```
./configure --enable-unicode && make && make install
```

Pro provedení `make install` musíte mít práva *root*. Po skončení překladu je ještě nutné spustit příkaz `ldconfig`.

Posledním krokem je nastavení překladu projektu v souboru `src/config.cfg` v kořenové složce projektu, v němž je třeba upravit proměnnou *DEBUG* (pokud byl překlad v *Release* verzi, pak nastavte hodnotu 0, jinak 1) a proměnnou *WX*, do níž uved'te příkaz, kterým se spouští `wx-config` zjišťující údaje pro překlad wxWidgets ve verzi 2.8. Obsahem této proměnné bude nejspíše `wx-config`.

Překlad projektu se provádí voláním příkazu `make` v kořenové složce projektu.

5.5.2 Windows

Pod operačním systémem Windows probíhá překlad překladačem *MinGW* (www.mingw.org), který je verzí *GNU C++* pro tento systém. Je nutné mít ho nainstalovaný, aby byl možný překlad nejen tohoto projektu, ale také knihovny *wxWidgets*.

MinGW nesmí být nainstalován ve složce obsahující mezery, proto je nejlepší instalovat přímo do kořenového adresáře.

Po nainstalování překladače je potřeba nastavit systémovou proměnnou *PATH*, aby bylo možné spouštět příkazy pro překlad v příkazovém řádku v kterékoli složce. Do proměnné přidáme na konec cestu ke složce `bin` uvnitř nainstalovaného překladače. Problém je s příkazem `make`, který má u *MinGW* název `mingw32-make`. Stačí ale soubor zkopírovat s novým jménem a je po problému.

Ze stránek www.wxwidgets.org nyní stáhneme verzi *wxWidgets* pro Windows, tedy archiv *wxMSW*. Doporučuji opět rozbalit do složky bez mezery v názvu, tedy nejlépe kořenový adresář. Po rozbalení přejdeme do složky `build/msw` v kořenové složce *wxWidgets*, v níž spustíme příkaz

```
make -f makefile.gcc UNICODE=1 BUILD=release
```

Součástí archivu *wxMSW* není program `wx-config`. Avšak pro Windows existuje verze také, najdete ji na stránkách `wxconfig.googlepages.com`. Tento program musí být opět spustitelný ze všech míst z příkazového řádku, a proto se jako nejlepší řešení jeví zkopírování programu do složky `bin` v překladači *MinGW*.

Pod operačním systémem Windows je nutné vytvořit dvě systémové proměnné, které využívá `wx-config` — *WXWIN* a *WXCFG*. *WXWIN* určuje úplnou cestu, v níž je knihovna wxWidgets nainstalována. *WXCFG* určuje způsob, jakým byly *wxWidgets* přeloženy. Jedná se v podstatě o složku, z níž budou načítány knihovny. V našem případě by se mělo jednat o složku `gcc_lib\mswu`, která by měla být umístěna i ve složce `lib` v kořenové složce *wxWidgets*.

Posledním krokem je opět nastavení překladu projektu v souboru `src/config.cfg` v kořenové složce projektu, v němž je třeba upravit proměnnou *DEBUG* (pokud byl překlad v *Release* verzi, pak nastavte hodnotu 0, jinak 1) a proměnnou *WX*, do níž uveďte příkaz, kterým se spouští `wx-config` zjišťující údaje pro překlad wxWidgets ve verzi 2.8. Obsahem této proměnné bude nejspíše `wx-config`.

Překlad projektu se provádí voláním příkazu `make win` v kořenové složce projektu.

Kapitola 6

Shrnutí práce

V této kapitole jsou diskutovány výhody a nevýhody aplikace a její další možná rozšíření.

6.1 Výhody

Mezi hlavní výhody patří jednoduchost při vytváření konečných automatů. Dalšími výhodami pak může být snadná konverze konečných automatů na speciální typy, která se snaží být také názorná, což rozhodně přispívá k rychlejšímu a snadnějšímu pochopení teorie převodů konečných automatů. Jednoduché ověřování správné funkce konečného automatu pomocí ilustrace činnosti určitě také najde dobré využití, stejně jako export do jazyka C.

6.2 Nevýhody

Odpůrcům grafického rozhraní může chybět podpora ovládání z příkazového řádku (terminálu). Program by zajisté našel uplatnění v situacích, kdy by fungoval jako nástroj pro práci s konečnými automaty bez grafického rozhraní. Pouze by na vstupu očekával konečný automat, nad nímž by provedl jistou operaci, a na vstup by uložil její výsledek. Například u převodů by byl na vstupu soubor s konečným automatem a výsledkem by byl převedený konečný automat. Nebo u simulace by konečného automatu by bylo výsledkem, jestli daný řetězec je nebo není součástí jazyka přijímaného konečným automatem. Za těchto okolností by mohl být i zjednodušen formát souborů konečných automatů, nebylo by třeba informací o grafickém rozložení. Úkolem však bylo vytvořit grafické rozhraní, což bylo splněno.

6.3 Možná rozšíření a dodělávky

Každého by jistě napadlo nějaké rozšíření, případně vylepšení určité funkce aplikace. Aplikace sama volá po opravení automatického rozmístění stavů a pravidel při konverzi z matematického na grafický model konečného automatu. Dále by mohl být vylepšen systém popisování převodů konečných automatů tak, že by byl rovnou vykreslován výsledný konečný automat. Ve spojení s vylepšeným automatickým rozmíst'ováním by byly převody jistě více názornější.

Novými funkcemi by pak mohla být návaznost s regulárními výrazy jako další teoretickou oblast regulárních jazyků, např. vytváření konečného automatu z regulárního výrazu nebo převod konečného automatu na regulární výraz.

Kapitola 7

Závěr

Cílem této práce bylo popsat významné celky vytvořeného demonstračního programu konverzí konečných automatů.

V první kapitole byly popsány způsoby implementace konečného automatu a jeho dva modely, které aplikace používá.

Druhá kapitola pojednávala o jednotlivých speciálních typech konečných automatů a implementacích algoritmů pro dosažení těchto typů.

Třetí kapitola se zabývala ilustrací činnosti konečných automatů a popsala dva módy ilustrace, které aplikace implementuje.

Další významnou částí aplikace je export do obrázku a do zdrojového kódu v jazyce C, který po přeložení simuluje činnost konečného automatu. Tato část je popsána ve čtvrté kapitole.

Pátá kapitola se věnovala několika bodům aplikace, které by nevydaly na celou kapitolu, přesto však bylo důležité je zmínit. Tato kapitola obsahovala popis souboru pro konečné automaty, třídy pro podporu vícejazyčnosti, myšlenky automatického rozmístění a její základní implementaci, systém příkazů grafického modelu konečného automatu a návod pro překlad projektu.

V šesté kapitole pak byly diskutovány přednosti a nedostatky praktické části projektu a jeho další možný vývoj.

Součástí některých kapitol byla i teorie, kterou bylo důležité uvést kvůli přesnému vyjádření daného problému.

Literatura

[1] Design Pattern (computer science) - Wikipedia, the free encyclopedia. [online], [cit. 2008-04-15].
URL http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

[2] Flex - a scanner generator. [online], [cit. 2008-04-25].
URL http://www.gnu.org/software/flex/manual/html_mono/flex.html

[3] Gansner, E. R.; Koutsofios, E.; North, S. C.; aj.: A Technique for Drawing Directed Graphs. [online], [cit. 2008-04-29].
URL http://www.graphviz.org/Documentation/TSE93.pdf

[4] JTC 1/SC 22/WG 14: ISO/IEC 9899:1999. [online], 1999, [cit. 2008-04-28].
URL http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf

[5] Kurz Formální jazyky a překladače, FIT VUT v Brně. [online], studijní materiály, [cit. 2008-04-15].
URL http://www.fit.vutbr.cz/study/courses/IFJ/

[6] Kurz Základy počítačové grafiky, FIT VUT v Brně. [online], studijní materiály, [cit. 2008-04-15].
URL http://www.fit.vutbr.cz/study/courses/IZG/

[7] Meduna, A.: *Automata and Languages: Theory and Applications*. London: Springer, 2000, ISBN 1-85233-074-0, 916 s.

[8] Smart, J.; Roebling, R.; Zeitlin, V.; aj.: wxWidgets 2.8.7: A portable C++ and Python GUI toolkit. [online], November, 2007, [cit. 2008-04-28].
URL http://docs.wxwidgets.org/stable/

[9] TinyXml. [online], [cit. 2008-04-28].
URL http://www.grinninglizard.com/tinyxmldocs/index.html

[10] Žára, J.: *Moderní počítačová grafika*. Brno: Computer Press, první vydání, 2004, ISBN 80-251-0454-0, 609 s.